Exploring how OpenMP can improve application performance

Dehui Yu
School of Computer Science
University of Ottawa, Ottawa, Canada
dyu105@uottawa.ca

Presentation outline

1. Introduction
2. Literature Review
3. Case Study one
4. Case Study two
5. Summary

Introduction

This part will focus on three questions:

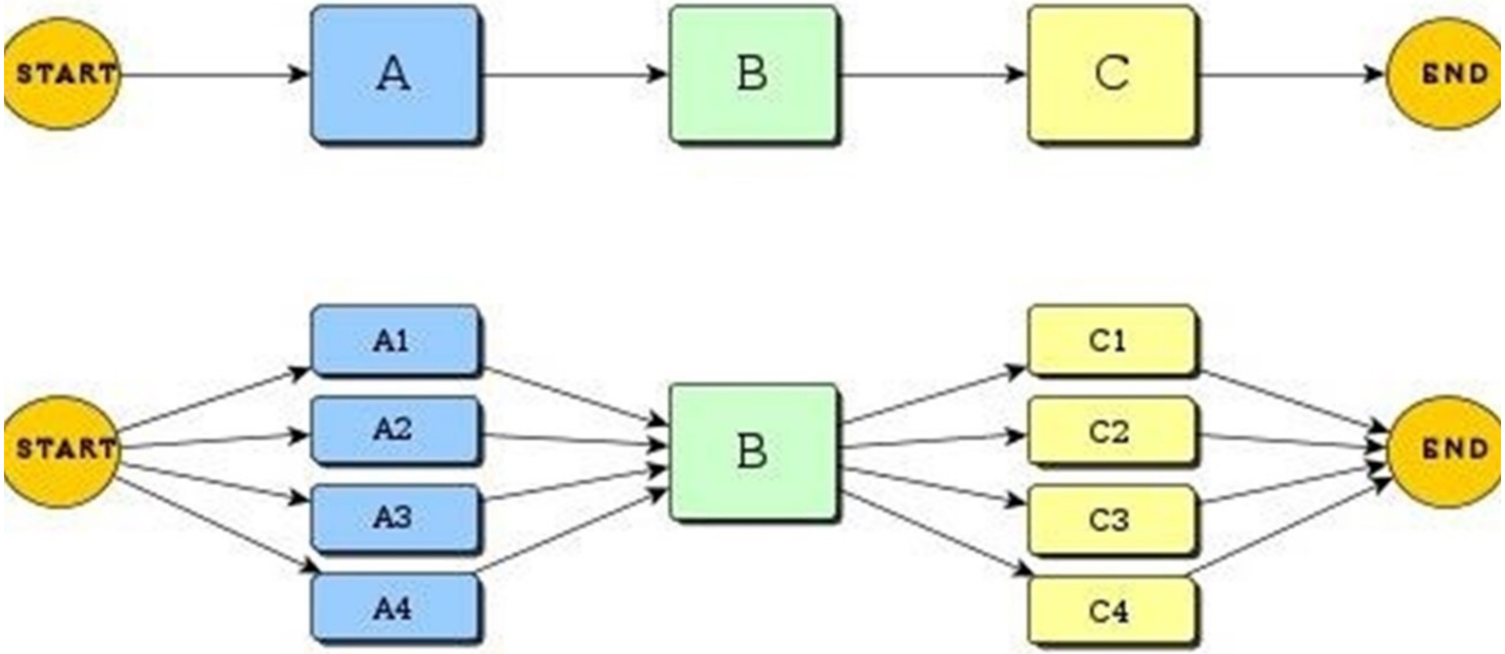
Question one: What is Parallel Programming?

Question two: What is OpenMP and MPI?

Question three: What is Monto Carlo?

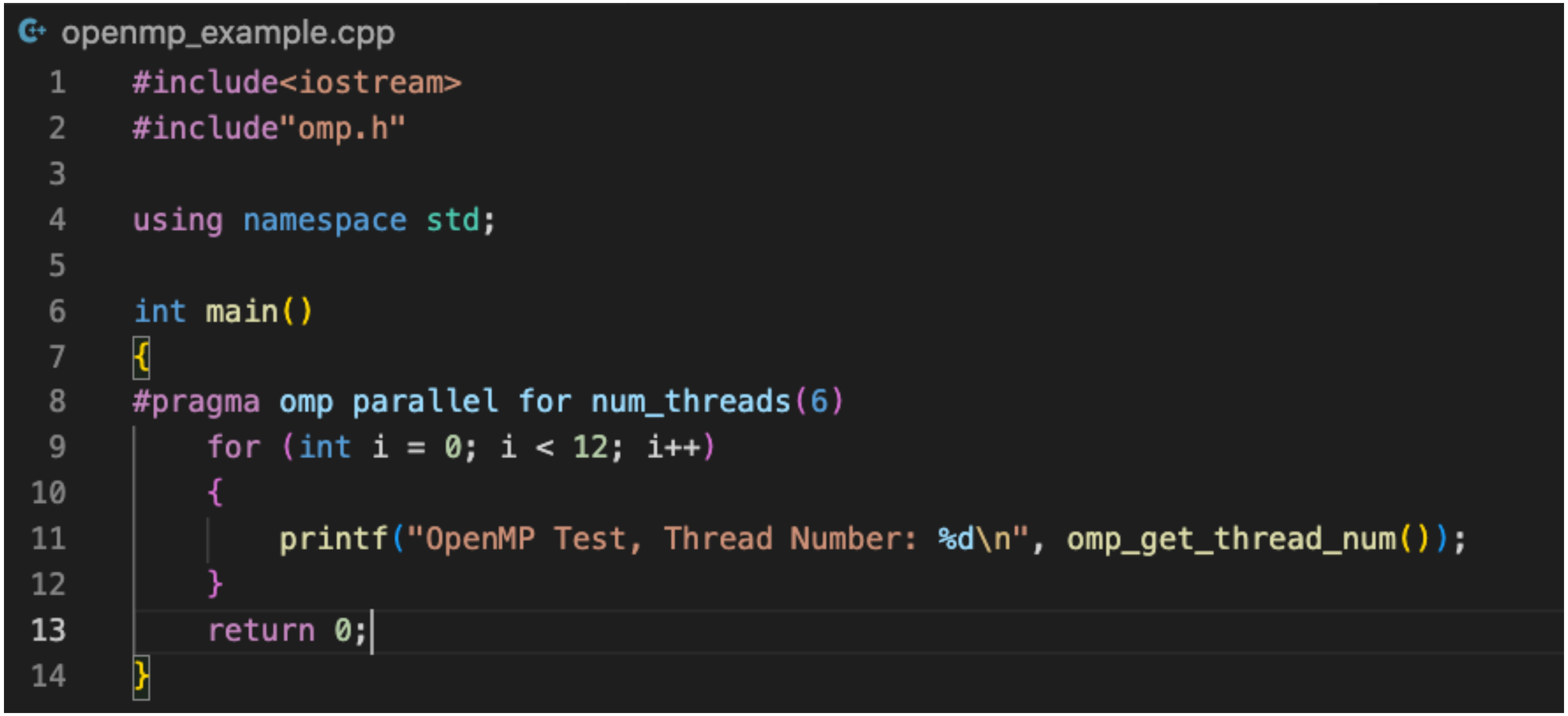
What is Parallel Programming?

Parallel programming, in simple terms, is the process of decomposing a problem into smaller tasks that can be executed at the same time using multiple compute resources.



What is OpenMP API?

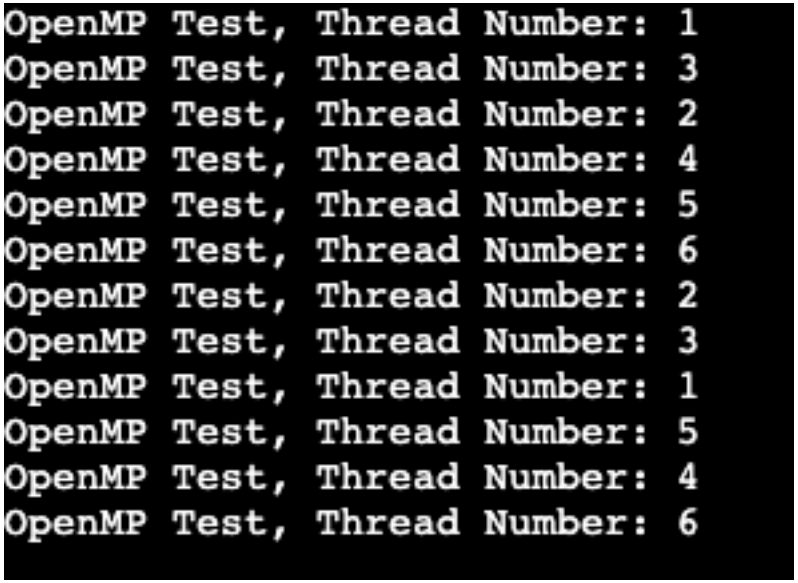
OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C, C++.



openmp_example.cpp

```cpp
#include<iostream>
#include"omp.h"

using namespace std;

int main()
{
#pragma omp parallel for num_threads(6)
    for (int i = 0; i < 12; i++)
    {
        printf("OpenMP Test, Thread Number: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Result

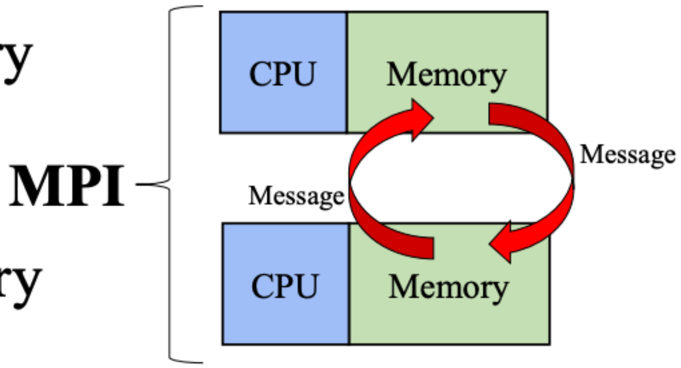


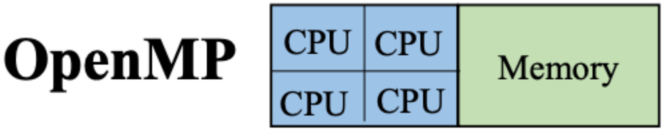
```
OpenMP Test, Thread Number: 1
OpenMP Test, Thread Number: 3
OpenMP Test, Thread Number: 2
OpenMP Test, Thread Number: 4
OpenMP Test, Thread Number: 5
OpenMP Test, Thread Number: 6
OpenMP Test, Thread Number: 2
OpenMP Test, Thread Number: 3
OpenMP Test, Thread Number: 1
OpenMP Test, Thread Number: 5
OpenMP Test, Thread Number: 4
OpenMP Test, Thread Number: 6
```

The above result specifies 6 threads, the iteration amount is 12, from the output can see that each thread is divided into 12/6=2 iteration amount.

What is MPI?

- MPI – Designed for distributed memory
  - Multiple systems
  - Send/receive messages
- OpenMP – Designed for shared memory
  - Single system with multiple cores
  - One thread/core sharing memory
- C, C++, and Fortran
- There are other options
  - Interpreted languages with multithreading
    - Python, R, matlab (have OpenMP & MPI underneath)
  - CUDA, OpenACC (GPUs)
  - Pthreads, Intel Cilk Plus (multithreading)
  - OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)





What is Monto Carlo?

- A powerful method that can be applied to otherwise intractable problems
- A game of chance devised so that the outcome from a large number of plays is the value of the quantity sought
- On computers random number generators let us play the game
- The game of chance can be a direct analog of the process being studied or artificial
- Different games can often be devised to solve the same problem
- The art of Monte Carlo is in devising a suitably efficient game.

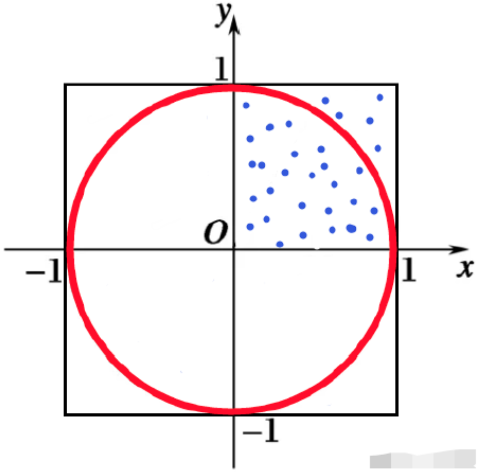
Monto Carlo – Calculating PI

We know that area of the square is $4r^2$ unit sq while that of circle is πr^2 The ratio of these two areas is as follows :

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

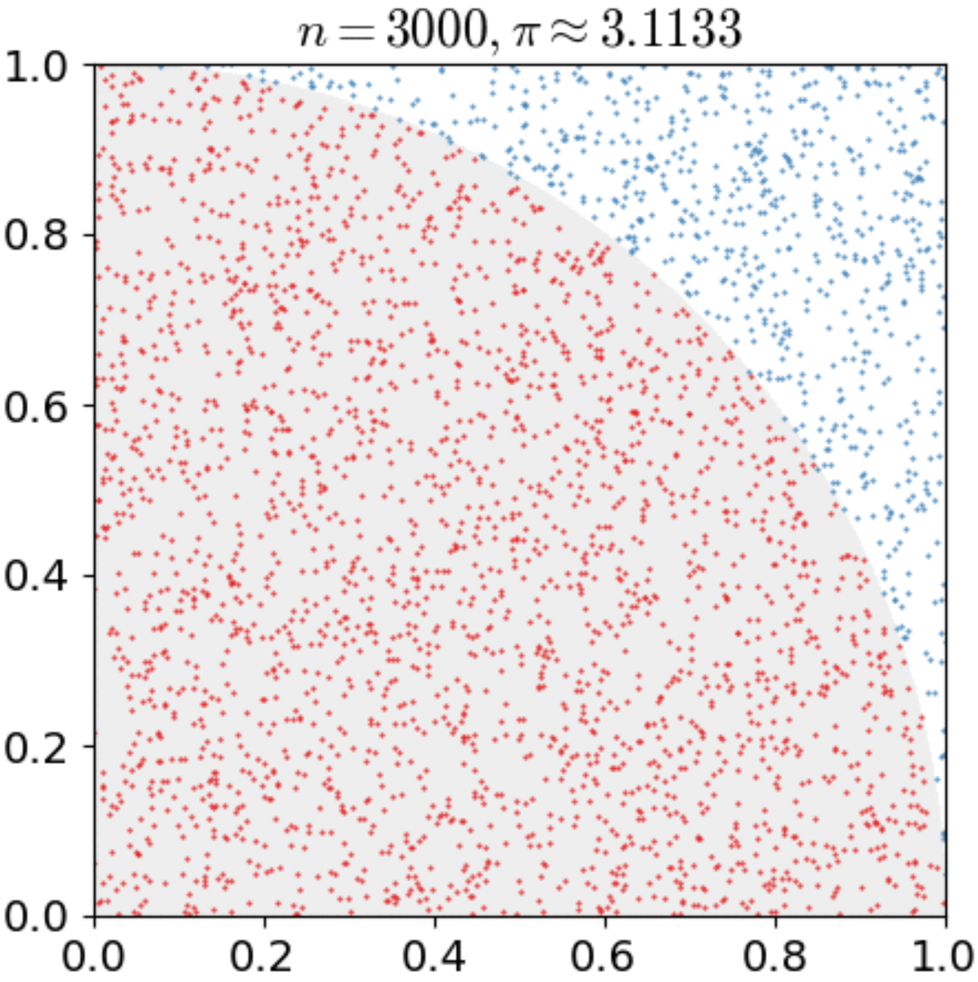
In other words, if there are a large number of generated points, the ratio can be presented as follows:

$$\frac{\pi}{4} = \frac{\text{no. of points generated inside the circle}}{\text{total no. of points generated or no. of points generated inside the square}}$$



The more the number of points scattered, the more accurate the PI will be.

Monto Carlo – Calculating PI



Literature Review

This part will focus on two questions:

Question one: Why OpenMP becomes the standard API for Parallel Programming?

Question two: Are there any performance improvement examples using OpenMP?

Why OpenMP becomes the standard API for Parallel Programming?

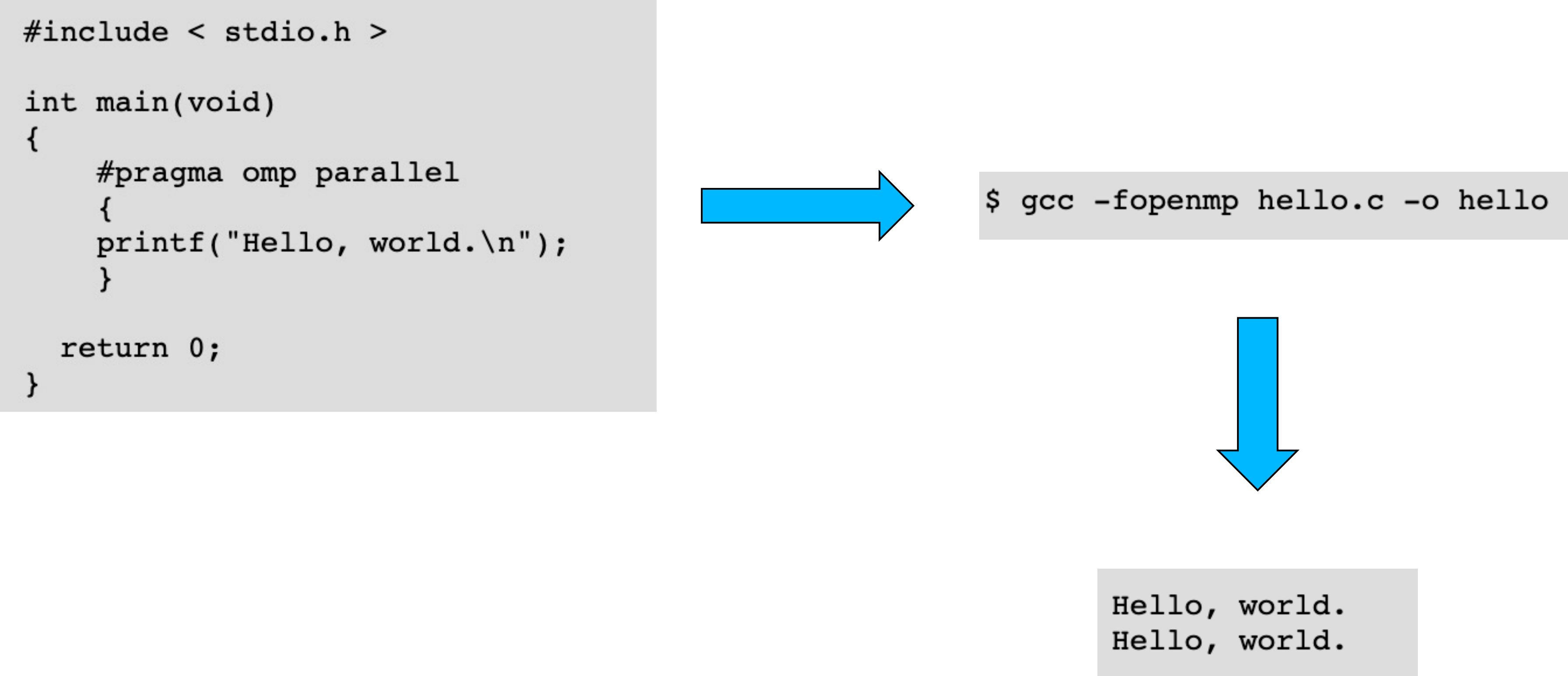
It support multiple languages.

OpenMP is designed for Fortran, C and C++. OpenMP can be supported by compilers that support one of Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, C11, C++11, and C++14, but the OpenMP specification does not introduce any constructs that require specific Fortran 90 or C++ features.

Cite: Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. "Using OpenMP." (2018).

Why OpenMP becomes the standard API for Parallel Programming?

It makes more efficient, and lower-level parallel code is possible



```
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

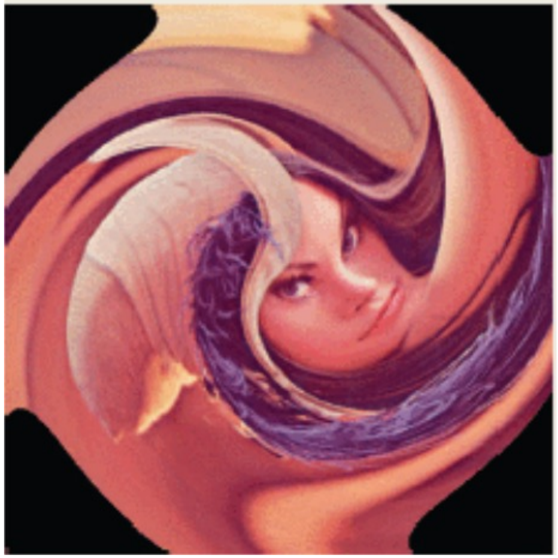
```
$ gcc -fopenmp hello.c -o hello
```

```
Hello, world.
Hello, world.
```

Cite: Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. "Using OpenMP." (2018).

Application example using OpenMP

(a)



G. Slabaugh, "Multicore Image Processing with OpenMP"

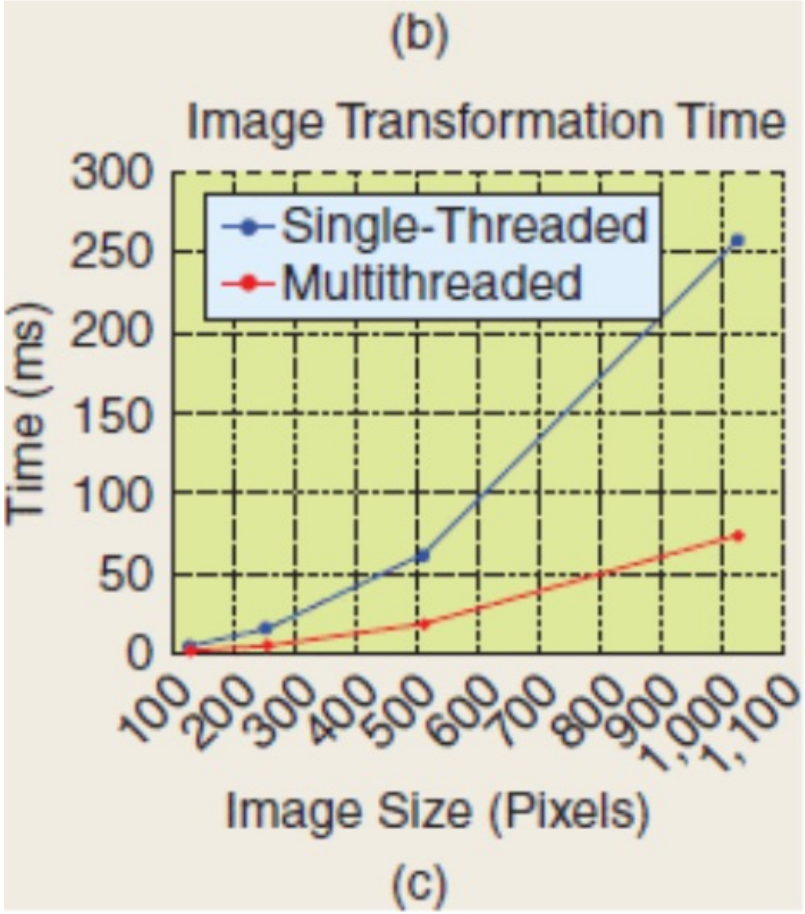
An image warp is a spatial transformation of an image and is commonly found in photo-editing software as well as registration algorithms. In this example, the author apply a "twist" transformation of the form.

$$
\begin{aligned}
x' &= (x - c_x)\cos\theta + (y - c_y)\sin\theta + c_x \\
y' &= -(y - c_y)\sin\theta \\
&\quad + (y - c_g)\cos\theta + c_y,
\end{aligned}
$$

On a 512 × 512 image, and using a Quad-core 2.4 GHz CPU Vs Single-core 2.4 GHz CPU

Multicore Image Processing with OpenMP

Application example using OpenMP



Multicore Image Processing with OpenMP

Problem Statement

1. By simply adding more threads, will the performance always be improved?

2. If the answer for question 2 is no, how many threads are the best for applications?

3. By running the same application using OpenMP and MPI, which one is faster?

Case Study

I made two OpenMP/MPI applications in C++ and test its performance

1. Calculating PI using OpenMP and MPI based on Monte Carlo

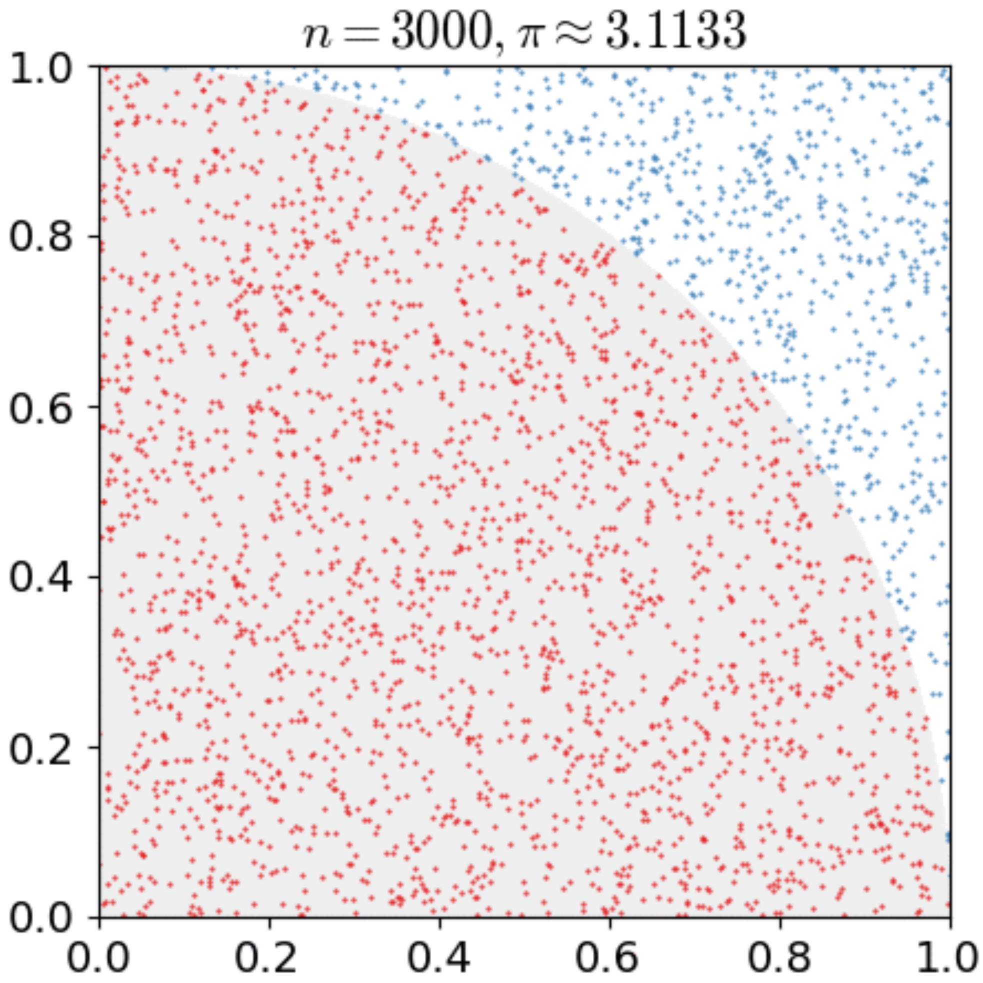
2. Matrix Multiplication using OpenMP

Experimental Environment

Platform: AWS EC2
Instance type: c4.4xlarge
16 vCPU
Operating system: Ubuntu 18.2
Max thread: 16
Price: 0.796 USD per Hour

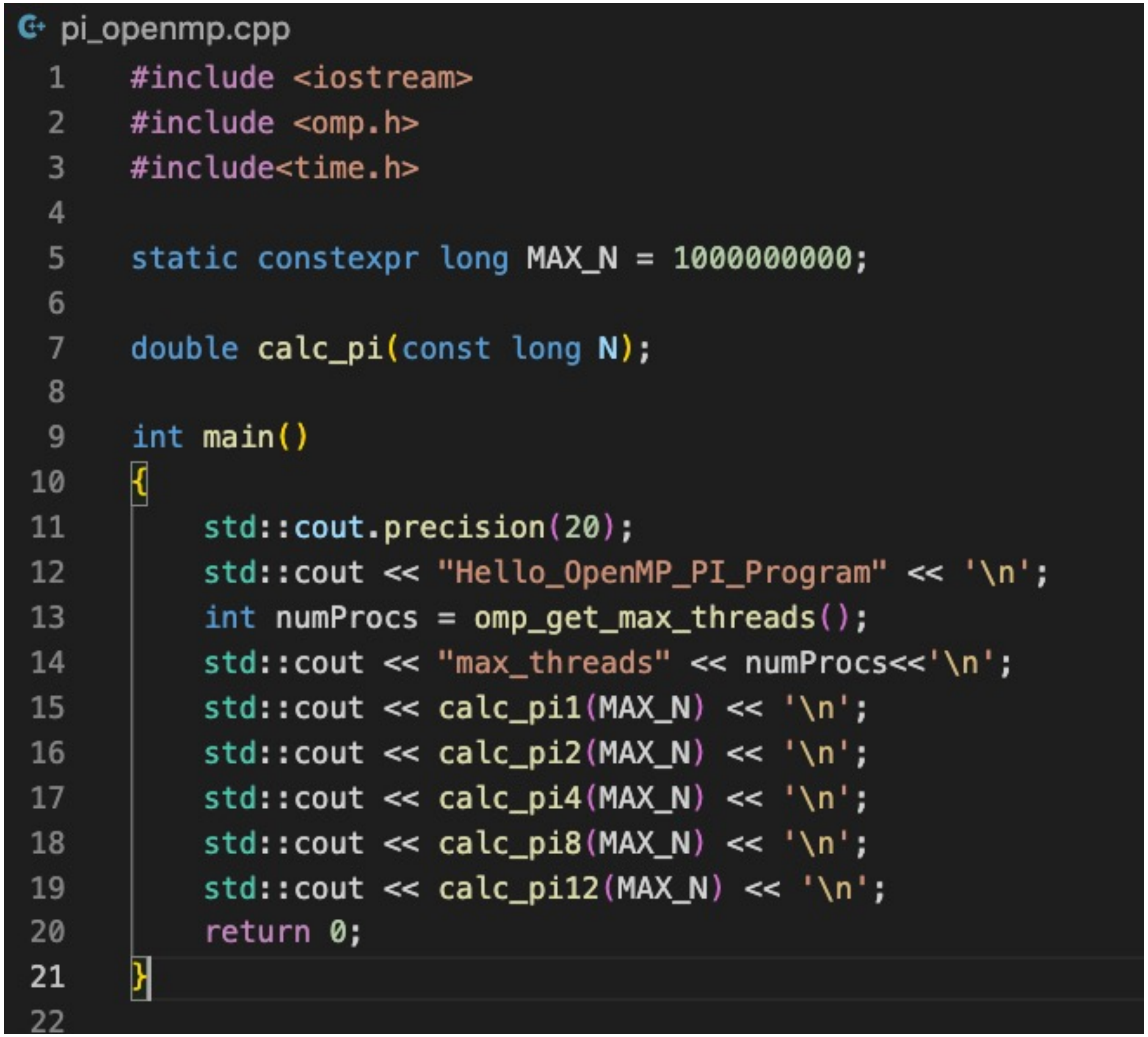
Amazon
EC2

Case Study One: Calculating PI using OpenMP based on Monte Carlo



We set:
N=100000000(10 million)

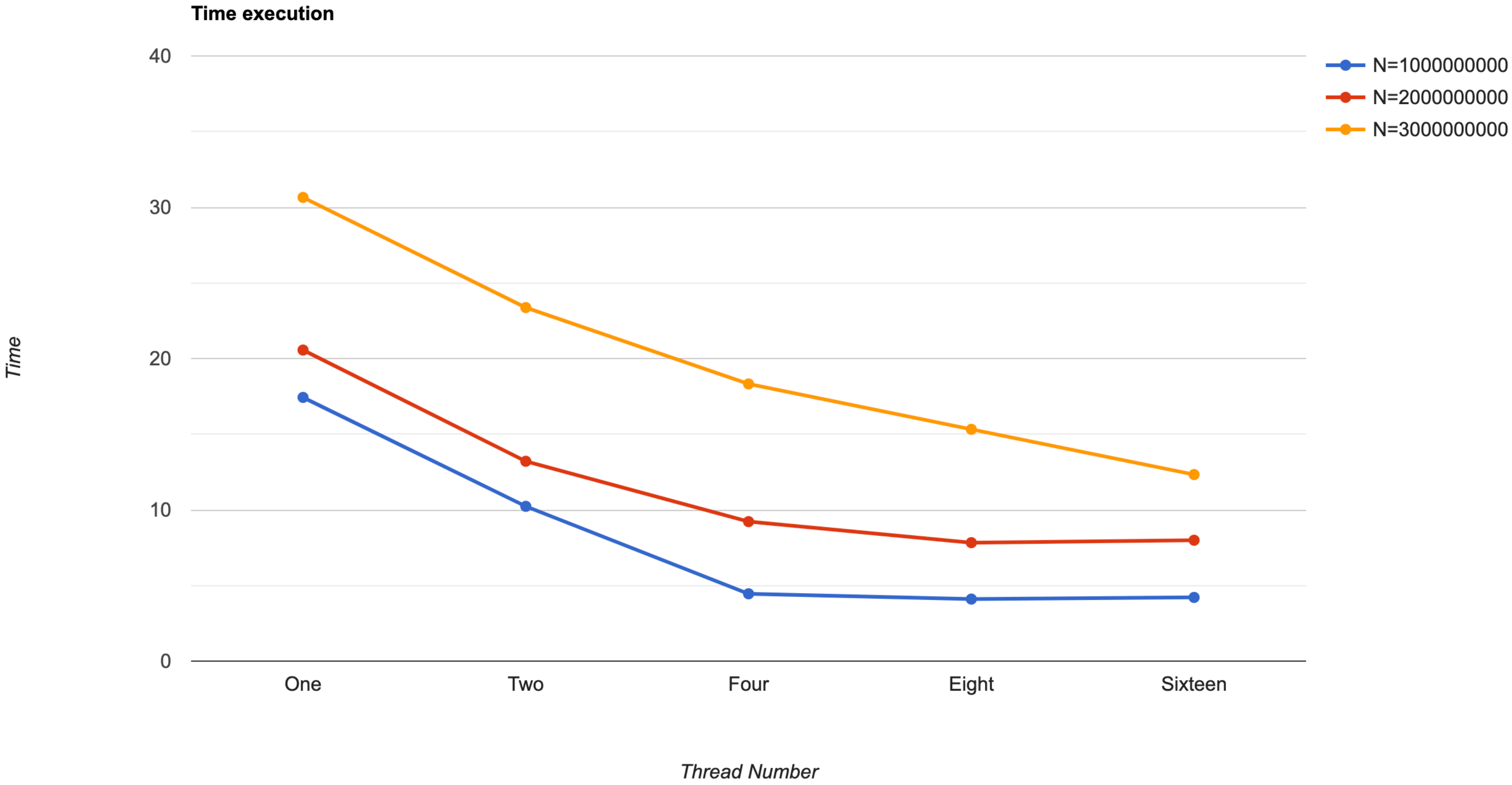
Source code – Case Study One



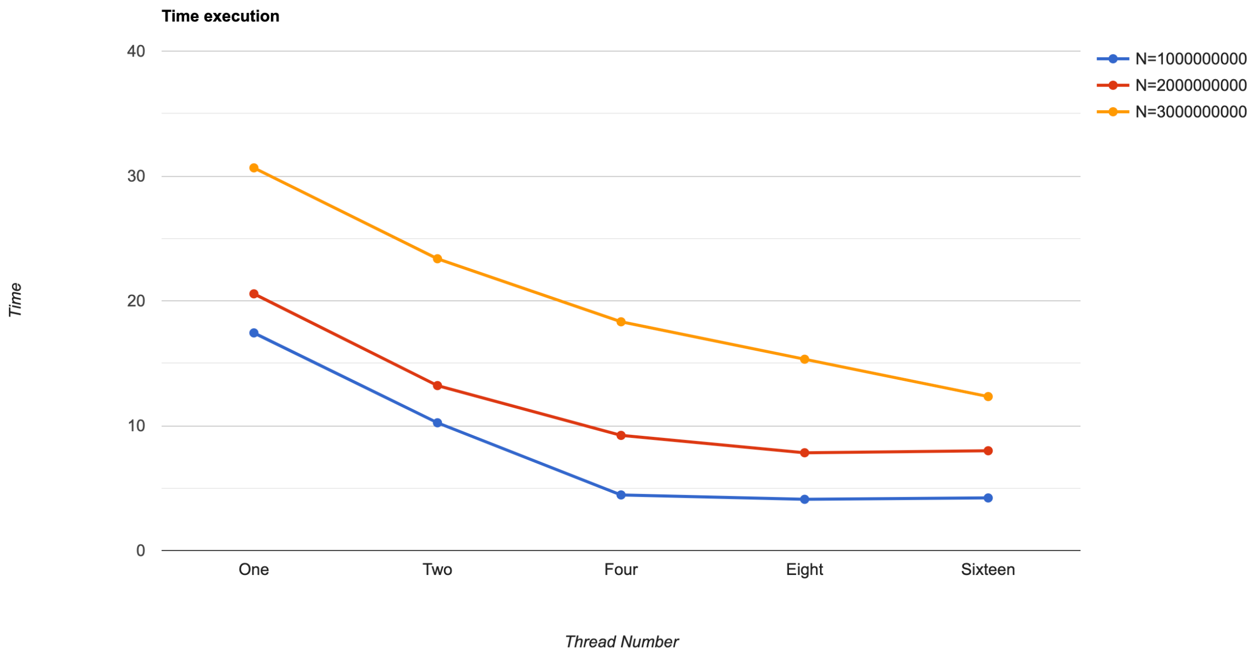
pi_openmp.cpp

```cpp
#include <iostream>
#include <omp.h>
#include<time.h>

static constexpr long MAX_N = 1000000000;

double calc_pi(const long N);

int main()
{
    std::cout.precision(20);
    std::cout << "Hello_OpenMP_PI_Program" << '\n';
    int numProcs = omp_get_max_threads();
    std::cout << "max_threads" << numProcs<<'\n';
    std::cout << calc_pi1(MAX_N) << '\n';
    std::cout << calc_pi2(MAX_N) << '\n';
    std::cout << calc_pi4(MAX_N) << '\n';
    std::cout << calc_pi8(MAX_N) << '\n';
    std::cout << calc_pi12(MAX_N) << '\n';
    return 0;
}

```

Results

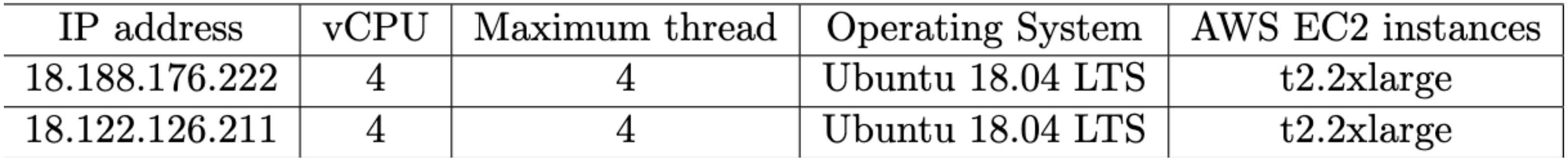

Time execution
40
30
20
10
0
Time
One
Two
Four
Eight
Sixteen
Thread Number
N=1000000000
N=2000000000
N=3000000000

Results

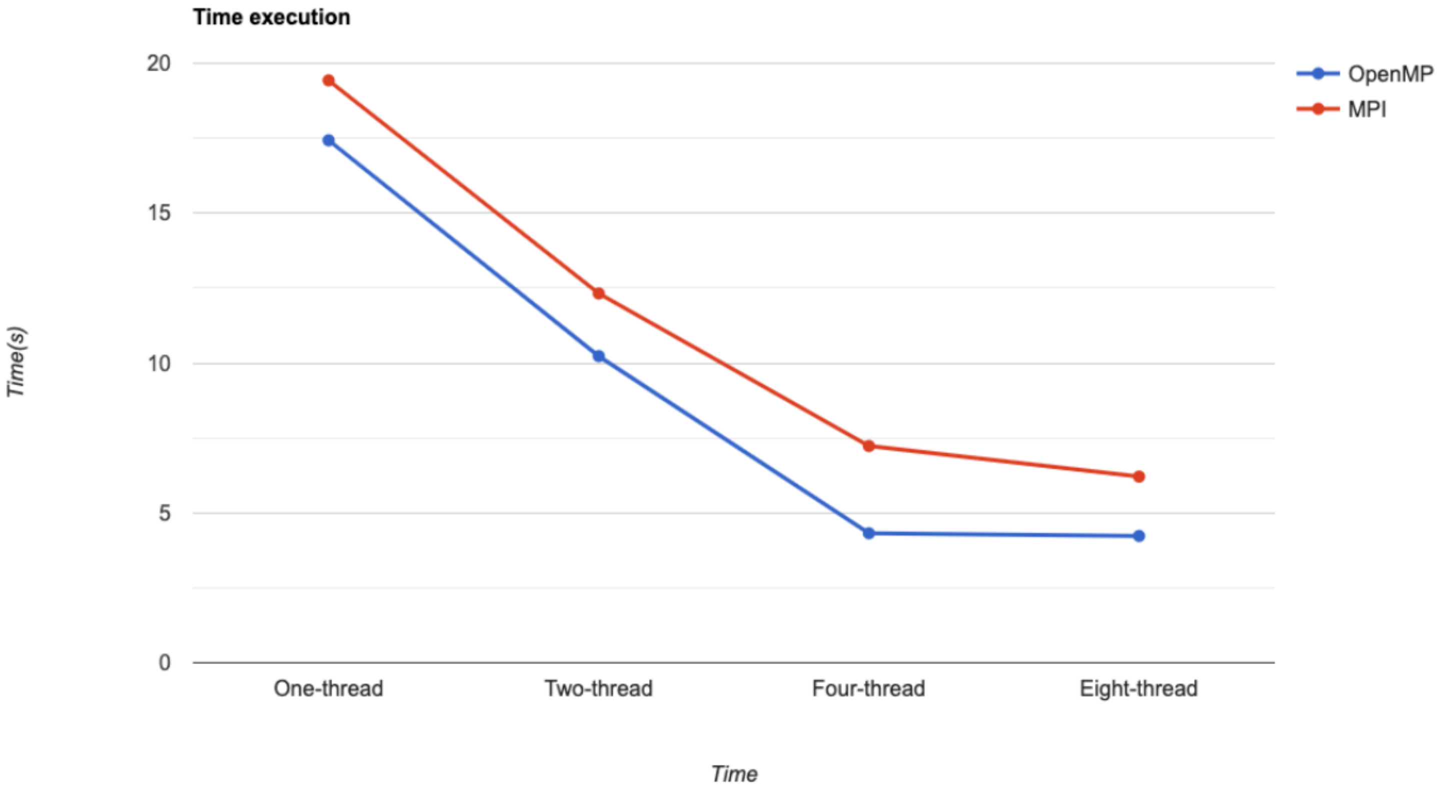


1. When N becomes larger, more threads can decrease running time.
2. By simply adding threads will not always improve application performance.
3. The best thread for this application is two thread or four thread.

Results: MPI – Calculating PI



IP address	vCPU	Maximum thread	Operating System	AWS EC2 instances
18.188.176.222	4	4	Ubuntu 18.04 LTS	t2.2xlarge
18.122.126.211	4	4	Ubuntu 18.04 LTS	t2.2xlarge



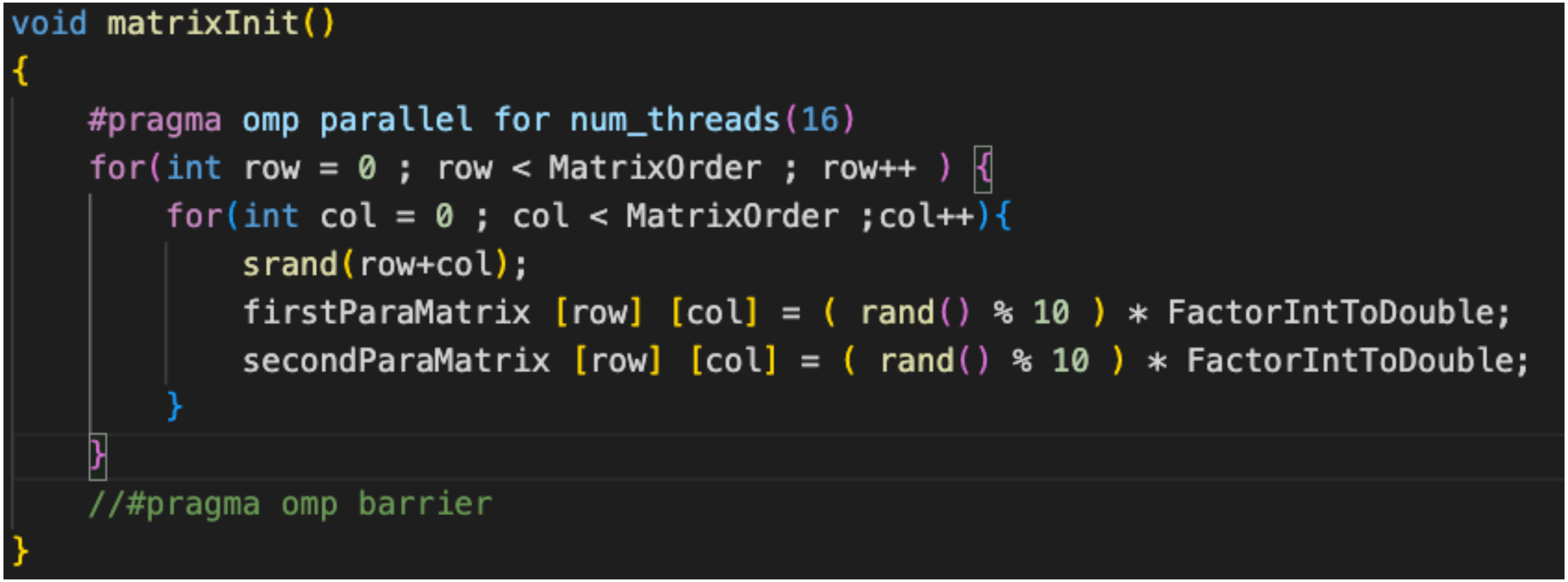
Case Study two: Matrix Multiplication using OpenMP

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} \quad ✓$$

2 Rows and 3 Columns

(1, 2, 3) • (7, 9, 11) = 1×7 + 2×9 + 3×11= 58
(1, 2, 3) • (8, 10, 12) = 1×8 + 2×10 + 3×12= 64
(4, 5, 6) • (7, 9, 11) = 4×7 + 5×9 + 6×11= 139
(4, 5, 6) • (8, 10, 12) = 4×8 + 5×10 + 6×12= 154

Source code



```
void matrixInit()
{
    #pragma omp parallel for num_threads(16)
    for(int row = 0 ; row < MatrixOrder ; row++ ) {
        for(int col = 0 ; col < MatrixOrder ;col++){
            srand(row+col);
            firstParaMatrix [row] [col] = ( rand() % 10 ) * FactorIntToDouble;
            secondParaMatrix [row] [col] = ( rand() % 10 ) * FactorIntToDouble;
        }
    }
    //#pragma omp barrier
}
```

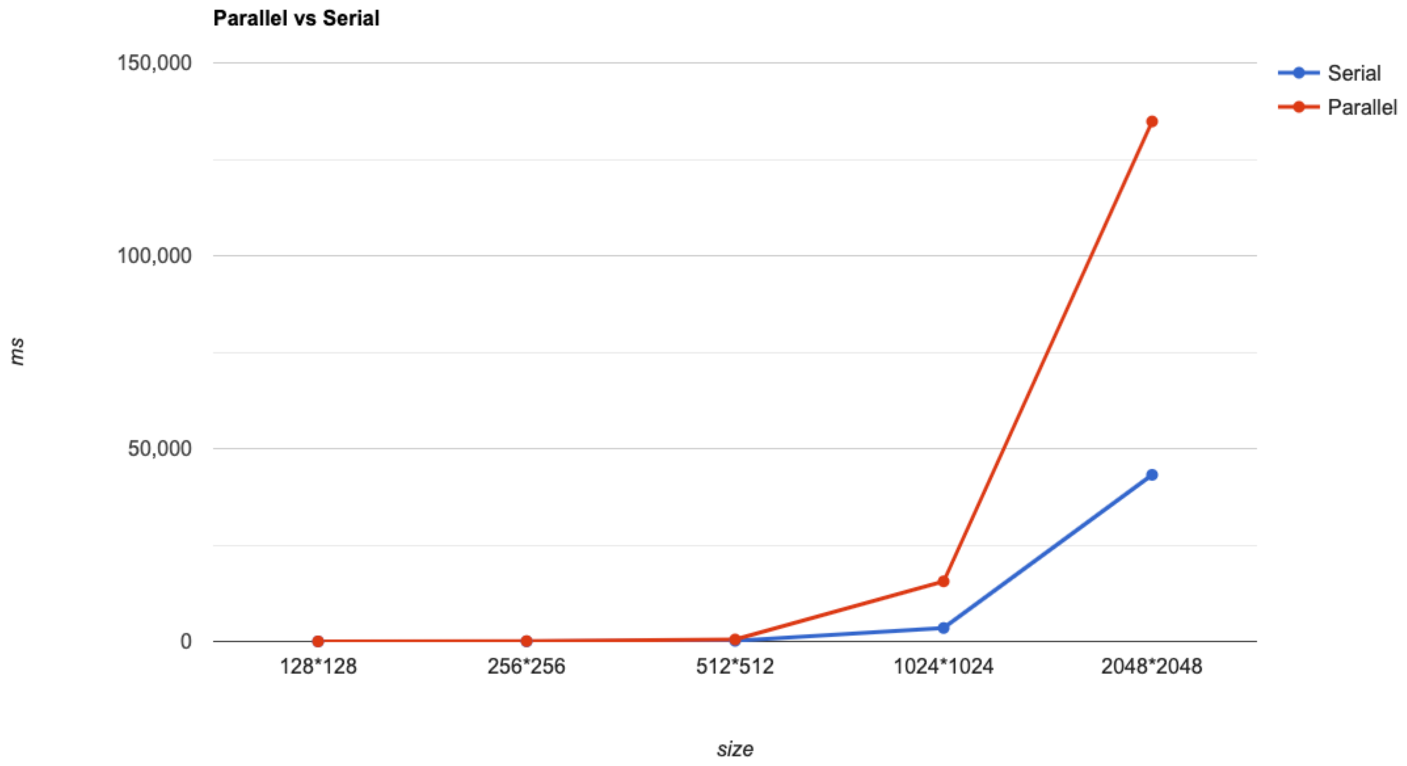
Case Study two: Matrix Multiplication using OpenMP

Platform: AWS EC2
Instance type: c4.4xlarge
16 vCPU
Max thread: 16
Price: 0.796 USD per Hour

Amazon
EC2

	128*128	256*256	512*512	1024*1024	2048*2048
Parallel (8 core)	2ms	31ms	164ms	3491ms	33203ms
Serial	16ms	100ms	516ms	15584ms	134818ms

Case Study two: Matrix Multiplication using OpenMP



As can be seen from the above chart, when the matrix size is small (order less than 500), the parallel programming has little gap with the serial programming. When the order reaches 1000 and 2000, the gap is obvious.

Summary

1. OpenMP has many benefits, and now it becomes the industry standard for Parallel Programming.
2. In our case study, I found by simply adding more threads will not always improve performance.
3. The best number of threads depend on application.
4. When running the same program, OpenMP running on shared memory is slightly quicker than using MPI on distributed memory.

Discussion/Future Research

1. Based on my introduction, do you really understand Monte Carlo algorithms?
2. What is the differences between MPI and OpenMP?
3. Does different operating system will have any impact on parallel computing using OpenMP?
4. Is the performance the same when compiling OpenMP applications with different compilers?

Partial references

[1] Eduard AyguadLe, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. IEEE Transactions on Parallel and Distributed Systems, 20(3):404–418, 2008.
[2] K Mani Chandy. Parallel program design. In Opportunities and Constraints of Parallel Computing, pages 21–24. Springer, 1989.
[3] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. Using OpenMP: portable shared memory parallel programming. MIT press, 2007.
[4] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. IEEE Computational Science and Engineering, 5(1):46–55, 1998.
[5] Thiago de Jesus Oliveira Duraes, Paulo Sergio Lopes de Souza, Guilherme Martins, Davi Jose Conte, Naylor Garcia Bachiega, and Sarita Mazzini Bruschi. Research on parallel computing teaching: state of the art and future directions. In 2020 IEEE Frontiers in Education Conference (FIE), pages 1–9, 2020.
[6] Xiongwei Fei, Kenli Li, Wangdong Yang, and Keqin Li. Cpu-gpu computing: Overview, optimization, and applications. Innovative Research and Applications in Next-Generation High Performance Computing, pages 159–193, 2016.
[7] Robert L Harrison. Introduction to monte carlo simulation. In AIP conference proceedings,
volume 1204, pages 17–21. American Institute of Physics, 2010.